Durante a aula, usamos sempre a sintaxe de criação de funções usando a palavra-chave `function`, apesar do JavaScript ter uma outra forma, mais curta e considerada mais "moderna" de se criar funções, a "arrow function" `=>`.

Então por que não utilizamos essa versão na criação dos nossos métodos?

Em um primeiro momento, todas as três formas de criação de função parecem funcionar de forma bem similar. Porém, a arrow function difere da `function` usual em alguns pontos, sendo o mais importante para nós nesse momento a questão do `this`.
Caso você precise relembrar como funcionam as três formas de se escrever funções em JavaScript, veja o tópico seguinte deste texto, "Relembrando os tipos de função" (logo abaixo).
A primeira diferença entre a declaração de função e as expressões de função é o *hoisting*. Mas, além do *hoisting*, existe outra diferença principal entre declaração de função e *arrow function*: ao contrário das funções normais, arrow functions herdam automaticamente o contexto de onde foram criadas e não têm seu próprio "contexto de invocação". Ou seja, não podem ser ligadas a contextos específicos com `this` e nem fazer uso dos métodos `bind()`, `call()` e `apply()`.
Arrow functions também não possuem a propriedade `prototype` e por isso não podem ser usadas como funções construtoras - assunto que veremos em seguida. Por este motivo, não usamos arrow functions em nenhum momento para a criação de métodos durante o curso.
Agradecemos ao aluno Rodolpho, que respondeu a essa dúvida sobre arrow functions no nosso fórum e nos deu a ideia de acrescentar este conteúdo.
Você pode consultar o MDN para saber mais sobre arrow functions.

Relembrando os tipos de função

A primeira versão, mais "clássica" e muito parecida com outras linguagens:

```
function soma(num1, num2) {
 return num1 + num2;
}
```

É a chamada “declaração de função”.

A segunda forma atribui a função a uma variável, funcionalidade que já não é tão comum em outras linguagens:

```
const soma = function(num1, num2) {
 return num1 + num2;
}
```

Chamamos essa forma de “expressão de função”.

A terceira e última forma é a *arrow function* ou “função de seta”, caracterizada pelo operador `=>`:

```
const soma = (num1, num2) => {
 return num1 + num2;
}
```

Ou, no caso de blocos de código com apenas uma linha, podemos omitir o `return` e as chaves `{}`:

```
const soma = (num1, num2) => num1 + num2;
```

Bem mais curto! A arrow function também é uma expressão de função.

E quais são as diferenças entre elas, além do `this`?
A primeira diferença é um pouco mais teórica. Funções criadas como *declaração* recebem um identificador (ou seja, um nome), e funções criadas como *expressão* são consideradas anônimas - estas funções são atribuídas a variáveis e é através dessas variáveis que conseguimos chamá-las e executá-las.
Na prática, a diferença se dá no contexto do carregamento do código. Declarações de função têm seu código lido antes da execução de qualquer instrução, pois o interpretador “puxa” para as primeiras linhas do arquivo todas as variáveis e todas as declarações de função, deixando seus códigos e dados “carregados” para aí sim começar a executar os códigos. A esta ação se dá o nome de **"hoisting"**, algo como “içar”, pois as funções e variáveis são “içadas” para o topo do arquivo e lidas primeiro.
Já as expressões de função, que são anônimas, não passam pelo processo de *hoisting* e têm seu conteúdo interpretado apenas no momento da execução. Assim, o interpretador (seja o Node.js ou um navegador) não consegue executar a função sem ter lido seu conteúdo antes.

Por exemplo, o código abaixo (uma declaração de função) executa normalmente:

```
console.log(soma(1, 1)) //2

//é possível executar a função antes de declará-la no código
function soma(num1, num2) {
 return num1 + num2;
}
```

Porém o código abaixo (uma expressão de função) não executa:

```
console.log(soma(1, 1)) //erro

//ReferenceError: Cannot access 'soma' before initialization
const soma = function(num1, num2) {
 return num1 + num2;
}
```

O mesmo erro acima ocorreria com uma arrow function. Faça o teste!

Agora que já relembramos os tipos de função e suas diferenças, podemos seguir em frente.